

電子動力学シミュレーションのステンシル計算最適化と メニーコアプロセッサへの実装

廣川 祐太^{1,a)} 朴 泰祐^{2,1} 佐藤 駿丞² 矢花 一浩^{2,3}

受付日 2016年4月22日, 採録日 2016年6月8日

概要: 近年, Intel Xeon Phi などメニーコアプロセッサを搭載した PC クラスタが運用されているが, 同プロセッサの性能特性から実アプリケーションにおいて高い性能を得るのは非常に困難である. 本研究では, 電子動力学シミュレータ ARTED での支配的な計算である波数空間と軌道に関して並列化された 3 次元実空間格子の 25 点ステンシル計算を, メニーコアプロセッサに対し最適化することを目的とする. まず, 元のターゲットシステムである京コンピュータ (SPARC64 VIIIfx) に対し最適化を行い, コンパイラによる自動ベクトル化を促進することで 14.94 GFLOPS から 27.2 GFLOPS に性能が向上した. この実装を用いて, メニーコアプロセッサの Intel Xeon Phi (Knights Corner) を対象に, 自動ベクトル化と Intrinsic を用いた手動ベクトル化による最適化を行った. 元実装が 30.06 GFLOPS であるのに対し, 手動ベクトル化実装で 224.45 GFLOPS と 20.9% のピーク演算性能比を達成した. また, 次世代プロセッサの Knights Landing への実装などについても考察する.

キーワード: 性能最適化, ステンシル計算, メニーコアプロセッサ, Intel Xeon Phi, SPARC64 VIIIfx

Optimization of Stencil Computation in Electron Dynamics Simulation and Implementation with Many-core Processor

YUTA HIROKAWA^{1,a)} TAISUKE BOKU^{2,1} SHUNSUKE A. SATO² KAZUHIRO YABANA^{2,3}

Received: April 22, 2016, Accepted: June 8, 2016

Abstract: Recently, PC clusters equipped with the many-core processors such as Intel Xeon Phi are actively operated. However, it is not easy to achieve high sustained performance on real applications because of special characteristics of this sort of processor. In this paper, we focus on an electron dynamics simulation code named ARTED in which 25-points 3-D stencil computation in real space grid parallelized over wave number space as well as orbitals is the core part of computation. First, we optimized its stencil computation to K computer (SPARC64 VIIIfx processor) that is the original target system of ARTED. As a result, the performance improved to 27.2 GFLOPS from 14.94 GFLOPS with automatic vectorization by compiler. Using this implementation, we applied explicit vectorization with intrinsics on its stencil computation part considering the features of current Intel Xeon Phi by Knights Corner architecture. As a result, we improved the sustained performance on a single Xeon Phi from poor original 30.06 GFLOPS to 224.45 GFLOPS on stencil computation which corresponds to approximately 20.9% of theoretical peak performance of single Xeon Phi. We also discuss on a future implementation on next generation of Knights Landing architecture.

Keywords: performance optimization, stencil computation, many-core processor, Intel Xeon Phi, SPARC64 VIIIfx

¹ 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

² 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba,
Tsukuba, Ibaraki 305-8577, Japan

³ 筑波大学大学院数理物質科学研究科
Graduate School of Pure and Applied Sciences, University
of Tsukuba, Tsukuba, Ibaraki 305-8571, Japan

a) hirokawa@hpcs.cs.tsukuba.ac.jp

1. 序論

今日、消費電力性能比からメニーコアプロセッサが注目され、約 60 の物理コアを持つ Intel Xeon Phi プロセッサを搭載したシステムが広く運用されている [1]. Xeon Phi プロセッサは、Many Integrated Cores (MIC) アーキテクチャに基づく x86 アーキテクチャベースのプロセッサで、現プロダクトは Knights Corner (KNC) と呼ばれる。KNC 上では、Linux カーネルが動作しているため、Xeon CPU で開発されたアプリケーションをコードの変更なく容易に実行可能だが、Xeon CPU に対してその性能特性は大きく異なる。ここで重要なのは、Xeon CPU 用コードの単純な移植では KNC の高い演算性能を得るのは非常に困難で、KNC に対する最適化が強く要求されることである [2], [3], [4]. 一方、現在、次世代 Xeon Phi として Knights Landing (KNL) が開発されている。KNC は PCI-Express で接続されるためホスト CPU を要求し、Xeon CPU と KNC のヘテロジニアスクラスタしか実装できなかった。KNL では、KNL 自体がホスト CPU として利用できるため、KNL のみを利用したクラスタが実装可能である。また、ポスト京コンピュータでも、メニーコア CPU を用いたシステムが想定されている。現在運用されている KNC クラスタは、これらのメニーコアシステムへ向けた評価と実装準備環境としても期待されている。

本研究では、KNC をターゲットに、電子動力学シミュレーションコードに現れる 3 次元 25 点ステンシル計算に対し、シングルコアレベルの最適化を行う。同シミュレーションは、空間的に周期境界条件を持つ電子軌道の時間発展を、時空間で差分化して解く計算が主要である。ステンシル計算は、周期境界条件を持つ深さ 4 の中心差分のため、本研究は、同様の計算パターンを持つアプリケーションにも寄与すると考えられる。また、本研究の成果が KNC 専用ではなく、今後のメインプロセッサとなることが予測される KNL など他のメニーコアプロセッサにおいても有効であることを考察する。

2. 対象とするステンシル計算

本研究では、筑波大学計算科学研究センターで開発中の電子動力学シミュレーションコード ARTED (Ab-initio Real-Time Electron Dynamics simulator) [5] において主要な計算時間を占める、電子軌道にハミルトニアン演算子を作用する際に現れるステンシル計算の最適化を行う。ARTED は、光科学分野の中心テーマである超短パルス光と物質の相互作用に対し、時間依存密度汎関数理論に基づき第一原理計算を行うアプリケーションである。パルス光が駆動する電子動力学を記述するとともに、パルス光の電磁場と電子の運動を、マルチスケール手法を用いて同時に記述することが可能である。電子動力学に関しては、電子

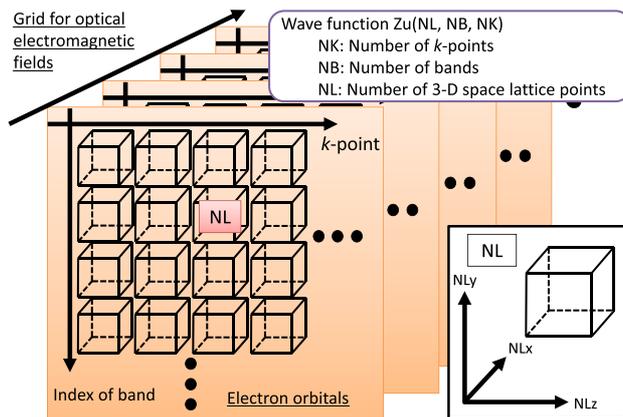


図 1 ARTED の計算領域のイメージ

Fig. 1 The schematic picture of ARTED computation fields.

軌道 (Electron orbitals) に対する時間依存コーン・シャム (TDKS, Time-Dependent Kohn-Sham) 方程式を実時間・実空間法を用いて解き、光電磁場 (Optical electromagnetic fields) に対しては有限差分時間領域 (FDTD, Finite Difference Time-Domain) 法を用いて解く [6]. これらのシミュレーションでは、TDKS 方程式に基づく電子軌道の時間発展計算に必要とされる、ハミルトニアン電子軌道に対する作用の計算 (以下、ハミルトニアン計算と呼ぶ) が、演算時間の大半を占めている。ハミルトニアン計算は、電子軌道に対するラプラシアン作用を含み、少ない空間格子点数で高い計算精度を得るために、深さ 4 の中心差分を用いたステンシル計算が行われる。図 1 に、電子軌道に対する配列 $Z_u(NL, NB, NK)$ のイメージを示す。電子軌道は、光電磁場の格子点ごとに用意され、プロット波数空間の格子点 (NK), バンド (NB), および差分計算が行われる 3 次元実空間格子点 (NL) の添字を持つ。光電磁場の各格子点で電子軌道の時間発展計算が行われ、光電磁場自身の計算コスト、および光電磁場格子点間の通信コストは低い。

ARTED における並列計算上の大きな特徴は、比較的サイズの小さい実空間格子点ではなく、光電磁場格子、電子軌道の波数空間格子とバンド添字に対して MPI と OpenMP でのハイブリッド並列を適用している点である。光電磁場格子および波数空間に対し MPI で並列分散を行い、分散した波数空間およびバンドに対し OpenMP で並列計算を行う。つまり、波数空間およびバンドはインデックスであり、このインデックスの数分、実空間の計算をシングルレッドで行う。波数空間を MPI で分散しているため、一般的にステンシル計算で問題となるプロセス間での袖領域通信が不要な代わりに全実空間格子点の総和通信を行う必要がある。しかしながら、実空間格子は波数空間格子および軌道に対しそれほど大きくはないため通信コストが低く、大規模並列システム向けとなっている。元のターゲットシステムは京コンピュータ [7] で、我々は KNC クラスタに対して同アプリケーションの最適化を行っている [8].

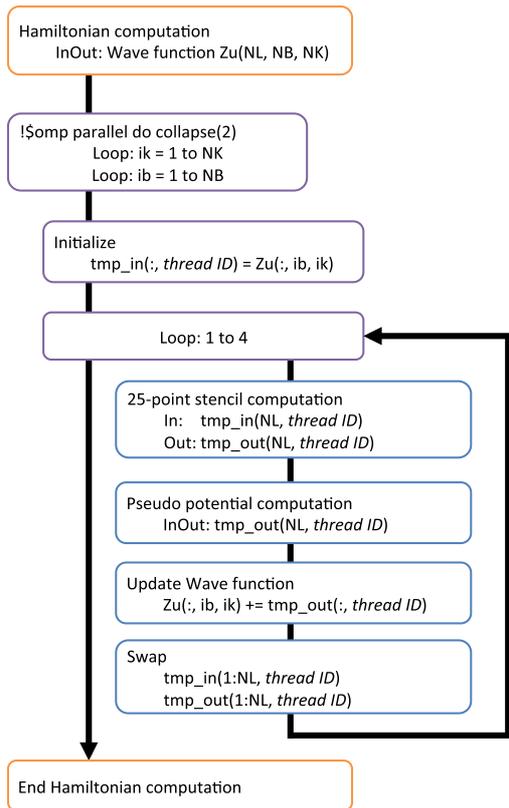


図 2 ハミルトニアン計算全体の流れ

Fig. 2 The computation flowchart of the Hamiltonian computation.

波数空間格子の規模は、シミュレーション対象や要求精度により異なるが、最終的には各実空間格子に対するステンスル計算の性能が全体性能を決定する。実空間格子のサイズは、シリコン結晶 (Si) を対象とした場合 (16, 16, 16), α クォーツ (SiO₂) を対象とした場合 (20, 36, 50) がアプリケーションの計算精度上の最小値となる。電子軌道配列は倍精度複素数で、周期境界条件による 25 点ステンスル計算が行われる。電子軌道の時間発展計算に 4 次のテイラー展開法が用いられているため、1 ステップの時間発展に 4 回のハミルトニアン計算が必要となる。ハミルトニアン計算の流れについて、図 2 に概略を示す。ハミルトニアン計算はステンスル計算と擬ポテンシャル計算で構成されている [5]。まず、各スレッドの一時計算領域 (tmp_in) に、計算する実空間をコピーする。コピーした後、ステンスル計算と擬ポテンシャル計算を行い、計算結果を格納した配列 (tmp_out) を用いて電子軌道配列 Zu の更新を行い、これを 4 回繰り返す。前述のとおり、波数空間並列で、各実空間を逐次的に計算するため OpenMP を用いて波数空間およびバンドのループ両方を並列化する。1 個の実空間の計算は OpenMP の 1 スレッドで行われるため、各スレッドは 1 回の時間発展で 4 回のステンスル計算が含まれるハミルトニアン計算を逐次的に行い、複数個の実空間の計算を行う。各実空間は独立であり、1 回の時間発展で行われる

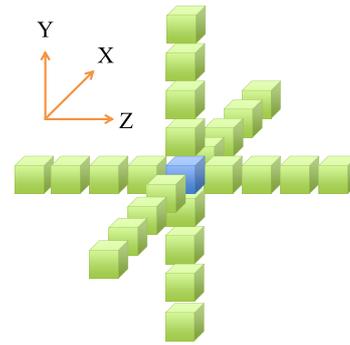


図 3 25 点ステンスル計算のメモリアクセスパターン

Fig. 3 The memory access pattern of the 25-points stencil computation.

```

integer, intent(in)      :: NL
integer, intent(in)      :: IDX(-4:4, 0:NL-1)
integer, intent(in)      :: IDY(-4:4, 0:NL-1)
integer, intent(in)      :: IDZ(-4:4, 0:NL-1)
real(8), intent(in)      :: A, B(0:NL-1)
real(8), intent(in)      :: Cx(4), Cy(4), Cz(4)
real(8), intent(in)      :: Dx(4), Dy(4), Dz(4)
complex(8), intent(in)   :: E(0:NL-1)
complex(8), intent(out)  :: F(0:NL-1)
complex(8), parameter    :: zI = (0.d0, 1.d0)
integer                  :: i
complex(8)               :: v(3), w(3)

do i=0, NL-1
  ! x computation
  v(1)=Cx(1)*(E(IDX(1,i))+E(IDX(-1,i)))&
  & +Cx(2)*(E(IDX(2,i))+E(IDX(-2,i)))&
  & +Cx(3)*(E(IDX(3,i))+E(IDX(-3,i)))&
  & +Cx(4)*(E(IDX(4,i))+E(IDX(-4,i)))
  w(1)=Dx(1)*(E(IDX(1,i))-E(IDX(-1,i)))&
  & +Dx(2)*(E(IDX(2,i))-E(IDX(-2,i)))&
  & +Dx(3)*(E(IDX(3,i))-E(IDX(-3,i)))&
  & +Dx(4)*(E(IDX(4,i))-E(IDX(-4,i)))
  ! y computation
  ...
  ! z computation
  ...
  ! store
  F(i) = B(i)*E(i) + A*E(i)      &
  & - 0.5d0*(v(1)+v(2)+v(3)) &
  & - zI*(w(1)+w(2)+w(3))
end do
  
```

図 4 ステンスル計算のオリジナル実装

Fig. 4 The original implementation of the stencil computation.

4 回のステンスル計算において OpenMP のスレッド同期または MPI による通信が発生しない。擬ポテンシャル計算はメモリバンド幅律速ではないため、実効メモリバンド幅に近い値を得るのは困難と予想される。しかしながら、ステンスル計算がハミルトニアン計算の 8 割以上を占めており [8]、本研究ではステンスル計算に対し最適化を行う。

図 3 に、25 点ステンスル計算のメモリアクセスパターンを示し、図 4 に、ARTED のステンスル計算のオリジナル

実装を示す。図 4 に示す計算は、図 2 の“25-point stencil computation”で行われる。ただし、Fortran では一般的に配列のインデックスが 1 で始まる 1-origin だが、最適化のため 0 始まりの 0-origin で受け取っている。Byte/FLOP 値は約 2.68 となり、メモリバンド幅律速な問題である。次に、図 4 のステンシル計算の詳細を説明する。A, B, C, D はすべて係数である。実空間格子は E, F に格納され、図 2 中の `tmp_in` および `tmp_out` に相当し、E に入力データが格納されており、F に計算結果を書き込む。実空間格子のメモリ配置は、Z 次元が連続、Y 次元が Z-stride, X 次元が ZY-stride となっている。本研究のステンシル計算は、配列 v, w に係数が異なる近傍点の加減算結果が各次元で格納されており、2 つのベクトル演算が行われている。周期境界を考慮したインデックス計算を省略するため、間接参照配列 `IDX, IDY, IDZ` にあらかじめ計算した近傍点のインデックスを格納している。必要な演算を洗い出すと、複素数の加減乗算、複素数のスカラ倍が必要となり、演算数は格子 1 点の更新に対し 158 FLOP である。本研究では、性能評価にシリコン結晶 (Si) を用いる。Si は、1 個のステンシル計算領域である実空間格子のサイズ ($NL = NL_x \times NL_y \times NL_z$) が $(NL_x, NL_y, NL_z) = (16, 16, 16)$ となる。MPI と OpenMP によって並列化される波数空間格子は、実アプリケーションでは 4^3 から 24^3 程度、バンドは 16 と設定するが、本研究では KNC のメモリ制限の関係でそれぞれ $8^3, 16$ と設定する。本研究の評価においては、サイズが 16^3 の逐次ステンシル計算を $8^3 \times 16$ 個と、非常に多くの個数を並列計算することが最も重要である。

本研究では、まず同コードが元のターゲットシステムである京コンピュータで十分な性能が得られているか検証する。その後 KNC に対し最適化を行い、性能評価と考察を行う。また、本研究中のすべての性能評価は 1 台のプロセッサ上で行う。

3. 京コンピュータ上の最適化

まず、図 4 の元実装が京コンピュータ上 (SPARC64 VIIIfx プロセッサ) で十分に最適化されているかを検証する [9]。表 1 に、プロセッサの諸元を示す。同プロセッサでは、128-bit SIMD 命令の HPC-ACE が提供されているが

表 1 SPARC64 VIIIfx の諸元

Table 1 Evaluation environment of SPARC64 VIIIfx.

Core	8
L1 Data Cache/Core	32 KB
L2 Cache	6 MB (shared)
GFLOPS	128
Memory Bandwidth	64 GB/s
Byte/FLOP	0.5
Compiler	Fujitsu K 1.2.0-19
Optimize Option	-O3 -Kfast, ocl, openmp

本研究の最適化対象は倍精度複素数計算のため、1 要素しか計算できない。そのため、本研究では HPC-ACE を用いた手動ベクトル化は行わず、コンパイラによる自動ベクトル化 (Compiler Vectorization) を促進する最適化を行う。

3.1 コードの問題点と修正

コンパイラによるベクトル化は、あくまでも記述されたコードを基に最適化を行うため、記述によってはベクトル演算が複雑化し期待した性能が得られていない可能性がある。図 4 のとおり、各次元で長さ 4 のベクトル演算が計 6 回行われている。近傍点のアクセスに用いている 4 Byte 整数の間接参照配列 `IDX, IDY, IDZ` により、コンパイラはメモリアクセスパターンを把握できず、メモリアクセスが非常に非効率的となっている。また、24 個の近傍点にアクセスするため、メモリから $96 \times NL$ Byte のインデックス値のロードが必要となり、メモリ帯域を非常に圧迫してしまう。評価に用いる空間格子のサイズは 16^3 で、データサイズにすると 64 KB となり、各点に対する係数 B も合わせて、各スレッドで 96 KB のデータをメモリからロードする。L1D キャッシュへのブロッキングを行うと、オーバヘッドが大きく性能低下につながったため、SPARC64 VIIIfx の計算には適用していない。L2 キャッシュはコア間共有だが、プロセッサ全体で 6 MB、コアあたり 768 KB が利用できる。必要な係数なども含んでも、1 個のステンシル計算で必要なデータは L2 キャッシュに収まっている。図 3 に示すとおり、メモリは Z 次元が連続となっているが、元実装では最もメモリ距離が遠い X 次元から計算しており、キャッシュの有効利用が行えていない可能性がある。そこで、Z, Y, X とキャッシュを利用しやすいように計算順序を変更した。

以上より、ステンシル計算の最適化を行ったコードを図 5 に示す。1 次元配列として確保している 3 次元実空間格子を、カーネル中では (NL_z, NL_y, NL_x) の 3 次元配列とした。この変更で、ステンシル計算時にインデックス計算が簡素化されるため、コンパイラがメモリアクセスパターンを把握しやすくなることが期待される。

本研究のステンシル計算は周期境界を持つため、インデックス計算には剰余が必要となる。実空間の各次元のサイズは、本研究では 2 のべき乗の 16 のため、論理積演算で代替可能かつ剰余演算よりも高速に求められるが、被除数が 2 のべき乗である必要がある。格子点を増やして 2 のべき乗にすることも可能だが、計算領域が肥大化しシミュレーションに制限が発生する可能性がある。そこで、各次元で剰余テーブル `modx, mody, modz` を用意し、剰余計算を省略した。剰余テーブルは、次元のサイズを N とすると $[0, N \times 2 + 4 - 1]$ の範囲の剰余計算結果が保存されている。本研究では、剰余テーブルは各次元で 144 Byte が必要で、これはすべてのインデックスを保管している間接参照

```

real(8),    intent(in)  :: B(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8), intent(in)  :: E(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8), intent(out):: F(0:NLz-1,0:NLy-1,0:NLx-1)
integer, intent(in)    :: modx(0:NLx*2+4-1)
integer, intent(in)    :: mody(0:NLy*2+4-1)
integer, intent(in)    :: modz(0:NLz*2+4-1)
integer      :: ix,iy,iz
complex(8)   :: v,w

#define IDX(dt) iz,iy,modx(ix+(dt)+NLx)
#define IDY(dt) iz,mody(iy+(dt)+NLy),ix
#define IDZ(dt) modz(iz+(dt)+NLz),iy,ix

do ix=0,NLx-1
do iy=0,NLy-1
do iz=0,NLz-1
! z computation
v=(Cz(1)*(E(IDX(1))+E(IDX(-1)))) &
& +Cz(2)*(E(IDX(2))+E(IDX(-2)))) &
& +Cz(3)*(E(IDX(3))+E(IDX(-3)))) &
& +Cz(4)*(E(IDX(4))+E(IDX(-4))))
...
! y computation
v=(Cy(1)*(E(IDY(1))+E(IDY(-1)))) ... ) + v
...
! x computation
...
! store
F(iz,iy,ix) = B(iz,iy,ix)*E(iz,iy,ix) &
& + A*(iz,iy,ix) - 0.5d0*v - zI*w
end do
end do
end do

```

図 5 コンパイラによる自動ベクトル化に最適化したステンシル計算の実装

Fig. 5 The implementation code of the stencil computation for compiler vectorization.

配列に対し非常に小さく、性能と可用性のバランスがとれている。

最後に、計算領域のサイズが2のべき乗のため、キャッシュスラッシングが多発する可能性がある。したがって、Y次元についてパディングを行い、キャッシュスラッシングを回避する。

3.2 最適化の効果

各最適化について、京コンピュータで提供されている詳細プロファイラを用い、8スレッド(コア)並列実行で評価を行った。ハミルトニアン計算には擬ポテンシャル計算が含まれるが、ステンシル計算に対し実行時間は比較的小さいため、本評価では計算を省略する。つまり、図2中で、擬ポテンシャル計算を省略したハミルトニアン計算を100反復行い、キャッシュミスや演算時間などの検証を行う。

まず、元実装でのロードストア命令に対するキャッシュミスを計測すると、L1Dミスが7.7%、L2ミスが0.03%となった。予測どおり、各実空間格子はL2キャッシュに収まっており、L1Dキャッシュミスの削減が重要となってい

表 2 L1D キャッシュミスおよびメモリプリフェッチ発行数
Table 2 The L1D cache miss rate, the number of L1D cache misses, and the number of memory prefetch instructions.

	元実装	最適化実装	+ パディング
ミス率 [%]	7.7	4.23	2.02
プリフェッチ数	1.48×10^7	1.41×10^{11}	1.41×10^{11}
ロードストア数	1.13×10^{12}	1.09×10^{12}	1.09×10^{12}

表 3 100 反復時のキャッシュメモリアクセス待ち時間 [sec]
Table 3 The cache access latency [sec] with 100 iterations.

実装	浮動小数点数	整数	トータル
元実装	50.67	40.11	90.78
最適化実装	43.86	4.63	48.49
+ パディング	8.71	4.80	13.51

表 4 100 反復時の演算待ち時間 [sec]
Table 4 The instruction latency [sec] with 100 iterations.

実装	浮動小数点数	整数	トータル
元実装	9.03	0.57	9.60
最適化実装	8.05	1.55	9.60
+ パディング	5.81	1.48	7.29

る。ロードストア命令比でのL1Dキャッシュミスを表2に示す。ここでは、図4の元実装、図5の最適化実装、最適化実装に対しパディングを行った際の結果を示している。最適化実装は元実装に対し、ロードストア命令数とL1Dキャッシュミスを削減できている。また、プリフェッチ命令の発行数が 10^7 オーダーから 10^{11} オーダーに増加しており、間接参照配列の除去によってプリフェッチが効果的に動作している。次に、キャッシュメモリへのアクセス待ち時間について、表3に示す。元実装では、100反復時に浮動小数点数と整数のアクセス待ち時間が合わせて90秒発生していた。最適化実装では、間接参照配列を用いず近傍点のインデックスを直接計算することによって整数のアクセス待ち時間が約1/10まで削減された。浮動小数点数は最適化実装でも依然として待ち時間が非常に長い。パディングによって1/5まで削減されている。ここで、演算待ち時間を表4に示すが、インデックスを直接計算することによって、整数演算の待ち時間が増加している。しかしながら、キャッシュメモリのアクセス待ち時間と合わせて100秒近いボトルネックが約20秒まで削減されており、間接参照のコストが非常に高いということが分かる。計算順序を非連続方向から計算した場合、ロードストア命令の発行数が1.25倍に増加し、キャッシュメモリのアクセス待ち時間が2倍、演算待ち時間が5倍近く増加した。非連続方向から計算したことにより、コンパイラは計算をパイプライン化できないという警告を出力しており、効率的なベクトル化および演算ができていなかったと考えられる。

最後に、演算性能について表5に示す。元実装では

表 5 SPARC64 VIIIfx でのステンシル計算性能

Table 5 The stencil computation performance on SPARC64 VIIIfx.

実装	GFLOPS	ピーク比 [%]
元実装	14.94	11.67
最適化実装	17.88	13.97
+ パディング	27.20	21.25

14.94 GFLOPS だったが、最適化およびパディングを行うことで 27.20 GFLOPS、ピーク性能比約 21%まで改善された。本研究では、図 5 の最適化した自動ベクトル化実装を用いて、KNC への最適化を行う。

4. Knights Corner への実装

4.1 自動ベクトル化コードの最適化

前述の最適化実装を用いて、KNC へのステンシル計算の最適化を行う。文献 [10] や [11] とした先行研究を参照すると、高レベル最適化を行ったとしてもピーク性能比 10%と、KNC において高い演算性能を得るのは非常に困難が予測される。本研究では、筑波大学計算科学研究センターの KNC クラスタ COMA を用いて KNC への最適化を行った [12]。KNC の諸元について、表 6 に示す。まず図 5 のコードで、元実装に対し KNC で性能が得られていることを確認する。KNC でのメモリのアラインメントは、実空間格子配列 $E(0:NLz-1, 0:NLy-1, 0:NLx-1)$ としたとき、 $E(0, :, :)$ が 64 Byte 境界にアラインされるように設定した。

まず、表 7 にコア内スレッド数による KNC でのステンシル計算性能への影響について示す。一般的に、ステンシル計算はメモリバンド幅速のため、物理コア数と OpenMP スレッド数を一致させることが望ましいとされている。しかしながら、本研究では小規模な実空間格子に対する逐次ステンシル計算を非常に多くの回数行うため、メモリバンド幅も重要だが同時処理数も重要であり、240 スレッド (コアあたり 4 スレッド) 時の実行性能が最も高い。本研究では、KNC は 60 コアを使用し、コアあたり 4 スレッド、計 240 スレッドですべての性能評価を行う。表 8 に、KNC でのステンシル計算性能を示す。図 4 の元実装は、ピーク性能比で 2.8%しか得られておらず、KNC でも間接参照のコストが非常に高い。自動ベクトル化実装は、図 5 を KNC で実行したときの性能で、元実装の約 3.6 倍に相当する 109.17 GFLOPS が得られた。同実装は、SPARC64 VIIIfx に対して最適化を行ったものだがシングルコアレベルの最適化のため、KNC においても有効と考えられる。この実装に対し、KNC へのさらなる最適化を行う。

図 5 では、近傍点のロードがベクトル演算と同時に進行しているため、一時配列に近傍点をロードした後に、ベク

表 6 COMA に搭載している Knights Corner の諸元

Table 6 Evaluation environment of Knights Corner on COMA.

Core	60 × 4 Thread
L1 Data Cache/Core	32 KB
L2 Cache	512 KB/Core
GFLOPS	1,074
Memory Bandwidth	352 GB/s
Byte/FLOP	0.3277
Compiler	Intel 15.0.2
Optimize Option	-O3 -openmp -restrict -ansi-alias -fno-alias -opt-assume-safe-padding

表 7 コア内スレッド数による性能への影響

Table 7 The performance impact with SMT threads.

スレッド数 (Threads/Core)	GFLOPS
60 (1 Threads/Core)	57.43
120 (2 Threads/Core)	83.55
180 (3 Threads/Core)	99.99
240 (4 Threads/Core)	109.17

表 8 Knights Corner での自動ベクトル化性能

Table 8 The compiler vectorization performance on Knights Corner.

実装	GFLOPS	ピーク比 [%]
元実装	30.06	2.80
自動ベクトル化実装	109.17	10.16
近傍点ロード最適化	125.81	11.71
non-temporal store	129.00	12.01
L1D ブロッキング	129.48	12.06
メモリパディング	130.44	12.15

表 9 倍精度浮動小数点数の Load/Gather 命令数

Table 9 The number of Load/Gather instructions with double precision values.

実装	Load	Gather
自動ベクトル化実装	568	96
近傍点ロード最適化	456	64

トル演算を行うように変更した。アセンブラコードから、Load および Gather 命令の出現数を表 9 に示す。KNC では、メモリアラインがとられていない状態でのロードに対して `loadunpackhi/lo` という 2 つの命令が用いられる。そのため、表での Load 命令数はアラインされたロード命令と合わせて 3 命令の総数を示している。アセンブラコードレベルでは、命令出現数が減少しており、処理内容が簡素化され効率が悪くなったかロード回数が減少したと考えられ、125.81 GFLOPS に性能が向上した。SPARC64 VIIIfx では、性能低下につながるため適用していない。

F には書き込みのみが行われ、書き込んだデータは計算中で利用されない。そこで、最内の Z 次元ループに `vector nontemporal(F)` ディレクティブを記述し、キャッシュを

表 10 L1D キャッシュミス数とヒット率

Table 10 The number of L1D cache misses and hit rate.

実装	ミス数	ヒット率 [%]
元実装	1.67×10^{11}	88.7
自動ベクトル化実装	7.83×10^{10}	85.8
近傍点ロード最適化	5.21×10^{10}	88.5
non-temporal store	4.76×10^{10}	89.6
L1D ブロッキング	4.79×10^{10}	89.0
メモリパディング	4.55×10^{10}	89.3

経由しない non-temporal store [13] とすることで、キャッシュメモリをさらに有効利用し 129 GFLOPS となった。SPARC64 VIIIfx では効果がなかったが、L1D に対して YX 次元に 2 次元ブロッキングを行うと、若干ながら性能が改善され 129.48 GFLOPS となった。スラッシングを回避するために SPARC64 VIIIfx と同様に Y 次元に対しメモリパディングを行ったが 130.44 GFLOPS と小さな改善にとどまった。Z 次元のパディングを行った場合、KNC ではベクトル化時にループサイズがベクトル長で割り切れなくなってしまう、アドレスが 64 Byte 境界からずれてしまうため、効率的なベクトル化が困難となる。

L1D キャッシュミス数とヒット率について、Intel VTune プロファイラ [14] での計測結果を表 10 に示す。元実装に対し、自動ベクトル化実装はミス数のオーダが 1 桁下がっている。近傍点のインデックスを直接計算したため、間接参照配列に使用していたキャッシュメモリをすべて近傍点に利用できたことが要因と考えられる。近傍点ロード最適化の効果が最も高く、L1D キャッシュミスが 2×10^{10} 程度減少した。他の最適化も適用することでミス数が減少しているが、L1D ブロッキング時には増加しており、効果が非常に小さいと考えられる。メモリパディングも行い、最終的な L1D キャッシュヒット率は 89.3% となった。

KNC では、ソフトウェアプリフェッチの適切な挿入が性能に大きく影響する可能性がある [13]。本研究では、ソフトウェアプリフェッチを `prefetch` デイレクティブやコンパイラオプション、手動ベクトル化実装では `Intrinsics` を用いて挿入し性能評価を行ったが、手動挿入による優位な結果は得られていない。

4.2 手動ベクトル化コードの実装と最適化

次に、KNC で提供されている 512-bit SIMD 命令の Initial Many-Core Instructions (IMCI) を用いた手動でのベクトル化 (Explicit Vectorization) を行う。Intrinsics は Fortran から利用することはできないため、本研究ではステンシル計算コードのみを C 言語の関数として抜き出し、手動ベクトル化実装を行う。本研究では、自動ベクトル化実装の最適化結果から、次に示す方法で手動ベクトル化を行う。まず、F に対して non-temporal store を用いるが、ストア先のアドレスが 64 Byte でアラインされていなければ

```
inline __m512d dcomplex_mul_c(__m512d a) {
    __m512i b, c, d = (__m512i)a;
    b = _mm512_set4_epi64(1LL<<63,0,1LL<<63,0);
    c = _mm512_shuffle_epi32(d, _MM_PERM_BADC);
    return (__m512d) _mm512_xor_si512(c,b);
}
```

図 6 倍精度複素数積 $(a, bi)(0, -i)$ を展開した実装Fig. 6 The expanded implementation of double complex multiplication $(a, bi)(0, -i)$.

ばならない。演算ベクトル長は倍精度複素数のため 4 となり、4 つの格子点を同時更新する。また、各次元で近傍点を先にロードし、その後演算を行う。前提条件として、連続なメモリアクセスとなる Z 次元について格子サイズを 4 の倍数 (ベクトル長) とし、最適化を行った。本研究では、主に以下の 3 点の最適化を行い、それぞれの効果について検証する。

- 倍精度複素数積の最適化
- 非アラインメモリアクセスの最適化
- 近傍点のインデックス計算のベクトル化

本研究の計算では、複素数が用いられているが、IMCI では Intel SSE/AVX と異なり、複素数積を高速に計算するための命令が提供されていない [15]。KNC で複素数積を行う際には、`masked` 命令を使って実部あるいは虚部のみを計算するの必要があり、ベクトル演算ユニットの潜在性能の半分しか利用できない。これは Intel コンパイラによる自動ベクトル化、Intrinsics を用いた手動ベクトル化の両方で問題となる。この問題から、KNC の複素数演算で高い性能を得るのは実数演算の場合よりも困難と考えられる。図 4 では、配列 w と定数 zI で積が必要となるが、定数積のため、式展開により演算を省略できる。複素数積 $(a, bi)(0, -i)$ を展開すると $(b, -ai)$ となり、(1) 実部と虚部を入れ替える、(2) 虚部の符号を反転させる、の 2 ステップで計算可能となる。図 6 に、式展開を行った IMCI 実装を示す。b は定数として扱われ、XOR を用いて虚部の実数の符号ビットを反転する。

連続方向である Z 次元の計算では、必ず非アラインメモリアクセスが発生するため、コンパイラは `Gather` 命令を発行しデータを集める。しかしながら、同命令はレイテンシが高く、性能上のボトルネックとなっている可能性が高い [16]。そこで、本研究では `Alignr` 命令を用いて `Gather` 命令を用いずに Z 次元の近傍点を集める。Alignr 命令は 2 つの 512-bit ベクトルを連結して 1,024-bit ベクトルとし、32-bit 単位で右論理シフトを行った後、下位 512-bit を返す命令である。Alignr 命令を用いた方法は文献 [17] などでも説明されているが、本研究では Alignr 命令を用いて周期境界条件に対するメモリアクセス最適化を行い、if 文による条件分岐コードを削除する。

ここで、メモリの連続方向と非連続方向での演算パター

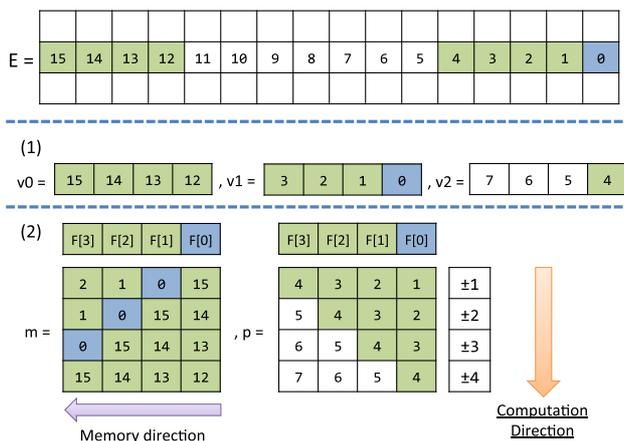


図 7 ベクトル演算での Z 次元のメモリアクセス

Fig. 7 The schematic picture of Z dimension memory access on vector operation.

ンの違いについて考える。連続方向では、Alignr 命令を用いて 1 点の更新に必要な近傍点を 1 つのベクトルに集めることが可能である。しかし、非連続方向では 1 回の Load 命令を用いて 4 点の更新に必要な近傍点のうち、±4 のいずれか 1 点を一度に取得できる。この場合、非連続方向はメモリに対して垂直方向に演算すれば、4 点を同時更新することができるが、連続方向では水平方向に演算する必要がある。SIMD での水平方向演算は IMCI には実装されておらず、AVX および SSE では加算が実装されているが演算コストが高い。そのため、連続方向でのメモリアクセスを、非連続方向と同じ垂直方向の演算となるように最適化を行う。

連続方向の Z 次元の非アラインメモリアクセスを、非連続方向と同じ演算パターンとなるように実装したイメージが図 7 となる。(1) まず、Load 命令を用いて更新対象の前の 4 点、更新対象の 4 点、その次の 4 点をそれぞれベクトル v_0, v_1, v_2 としてロードし、(2) Alignr 命令と v_0, v_1, v_2 を用いて、各点の更新に必要な近傍点ベクトルを生成する。このとき、更新点 ±4 でそれぞれ 1 つの行ベクトルを構成する。そのため、負方向と正方向の近傍点で 4×4 の正方行列 m および p が生成される。たとえば、更新点 $F[0]$ では、近傍点 15-12 および 1-4 が必要となる。生成した正方行列を見ると、非連続方向でのメモリパターンと同様に、必要な近傍点は 1 列に揃っており、ベクトル演算はメモリに対して垂直方向に演算を行うだけでよく、水平方向への演算が必要ない。このメモリアクセスを、Intrinsics を用いて実装すると図 8 となる。Alignr 命令の 32-bit シフト回数は即値で、コンパイル時には値が確定している必要がある。

近傍点は、各方向に ±4 点あり、各次元で 8 点、計 24 点となる。Z 次元は前述の最適化を行っているため、Y および X 次元の 16 点のインデックスを求める必要がある。インデックスは 4Byte 整数の表現可能範囲で十分足りるため、16 点のインデックスを 512-bit ベクトル演算でまとめて求めることができる。本研究では、Y および X 次元の近

```

/* Stencil computation loop */
for(ix = 0 ; ix < NLx ; ++ix) {
  for(iy = 0 ; iy < NLy ; ++iy) {
    double complex const* e = E[ix*NLy*NLz + iy*NLz];
    ...
    for(iz = 0 ; iz < NLz ; iz += 4) {
      __m512i v0, v1, v2;
      __m512d m[4], p[4];

      /* (1) */
      v0 = __mm512_load_epi64(e + modz[iz-4+NLz]);
      v1 = __mm512_load_epi64(e + iz);
      v2 = __mm512_load_epi64(e + modz[iz+4+NLz]);

      /* (2) */
      m[0] = (__m512d)_mm512_alignr_epi32(v1,v0,12);
      m[1] = (__m512d)_mm512_alignr_epi32(v1,v0, 8);
      m[2] = (__m512d)_mm512_alignr_epi32(v1,v0, 4);
      m[3] = (__m512d)v0;
      p[0] = (__m512d)_mm512_alignr_epi32(v2,v1, 4);
      p[1] = (__m512d)_mm512_alignr_epi32(v2,v1, 8);
      p[2] = (__m512d)_mm512_alignr_epi32(v2,v1,12);
      p[3] = (__m512d)v2;

      /* Z-dir.computation */
      for(n = 0 ; n < 4 ; ++n) {
        __m512d a = __mm512_add_pd(p[n], m[n]);
        __m512d b = __mm512_sub_pd(p[n], m[n]);
        v = __mm512_fmadd_pd(C[n], a, v);
        w = __mm512_fmadd_pd(D[n], b, w);
      }
      /* Y-dir. computation */
      /* X-dir. computation */
      /* non-temporal store to F */
    } } }

```

図 8 Z 次元のメモリアクセスを Intrinsics で実装したコード

Fig. 8 The implementation code of the Z dimension memory access with Intrinsics.

表 11 手動ベクトル化コードの最適化の効果

Table 11 The performance impact of the several optimization for explicit vectorization.

実装	GFLOPS	ピーク比 [%]
初期実装	113.65	10.58
複素数積最適化	114.89	10.70
非アラインアクセス最適化 (A)	179.39	16.70
インデックス計算ベクトル化 (B)	142.74	13.29
(A) + (B)	221.38	20.61
全最適化適用時	224.45	20.90

傍点のインデックス計算をまとめて一度ベクトル演算で求め、スカラでのインデックス計算を極力少なくした。Z 次元の計算では、前述の最適化でメモリから 512-bit ベクトル 3 本のみをロードするため、図 8 に示すようにスカラでインデックス計算を行っている。

手動ベクトル化実装に対し、各最適化を個別に適用した結果を表 11 に示す。単純に手動ベクトル化を行った初期実装では 113.65 GFLOPS となり、自動ベクトル化よりも

性能が低い。L1D キャッシュミスはすべての最適化で約 3.75×10^{10} 、ヒット率は92%程度となり、自動ベクトル化実装に対し良好な結果となっている。

初期実装に対し複素数積展開を行うと、114.89 GFLOPS と若干ながら性能が向上し、複素数積を計算するコストが非常に小さいといえる。本研究では、各点の更新に対し複素数積を一度しか行わず、演算数が少ないため、複素数積が非常に多く必要となる計算では注意が必要となる。非アラインメモリアクセス最適化は、179.39 GFLOPS と本研究の最適化の中で最も高い効果が得られ、Gather 命令が大きなボトルネックとなっていることが推測される。インデックス計算をベクトル化することで142.74 GFLOPS の性能が得られ、インデックスをベクトル演算で求める利点があることが分かった。さらに、非アラインメモリアクセスの最適化と、インデックス計算のベクトル化の2つを組み合わせ、221.38 GFLOPS を達成した。初期実装に対し約107 GFLOPS の性能向上が得られており、2つの性能の増分が約95 GFLOPS で、10 GFLOPS ほど追加の性能向上が得られている。命令数は変わっておらず、組合せによってレジスタの有効利用や計算の待ち時間が減少しさらなる性能向上につながったと考えられる。3つの最適化すべてを適用することで、初期実装に対し1.97 倍の224.45 GFLOPS とピーク性能比20.9%を達成した。

4.3 周期境界条件に対する最適化

本研究のステンシル計算は、周期境界のためキャッシュミスが発生しやすく、性能上不利なメモリアクセスが発生しやすい。既存研究 [8] においても、全次元を拡張し周期境界領域をコピーすることで、ステンシル計算中の全メモリアクセスを線形アクセスにする最適化を行い、性能向上を確認している。しかしながら拡張によって生じた前処理、周期境界領域へのデータコピーのコストが非常に高く、ハミルトニアン計算全体では性能が低下していた。そこで、本研究では連続方向のZ次元のみの拡張を考える。これにより、少なくともZ次元については線形アクセスとなり、より効率の高いベクトル演算が期待できる。

本研究のステンシル計算は、1個の実空間格子に対し4回行われるため、各スレッドが計算用の一次領域にコピーした状態で、ステンシル計算を行っている。ここでは、この一次領域のサイズを元の $E(0:NLz-1, 0:NLy-1, 0:NLx-1)$ から、 $E(-4:NLz+4-1, 0:NLy-1, 0:NLx-1)$ と拡張し、連続方向について線形アクセスとなるように最適化を行った。この最適化によって、各ステンシル計算の直前に $E(-4:-1, :, :)$ と $E(NLz:NLz+4-1, :, :)$ の領域に対しデータコピーが必要となるが、既存研究に比べコピーコストは低いことが期待される。

自動ベクトル化および手動ベクトル化の最適化実装に対し、Z次元のみ周期境界領域を拡張した場合の効果につい

表 12 周期境界領域拡張の効果

Table 12 The performance impact of extended periodic boundary area.

実装	Compiler Vec.	Explicit Vec.
従来 [GFLOPS]	130.44	224.45
Z次元拡張 [GFLOPS]	165.63	220.66
ハミルトニアン計算相対性能	1.03	0.93

て、表 12 に示す。以後、同最適化をZ次元拡張とする。自動ベクトル化実装では30 GFLOPS 以上の性能向上につながったが、手動ベクトル化実装では性能が低下した。本研究の実装がKNCに対し非常に最適化されており、同最適化の効果が小さくなっていると推測される。また、4行目には擬ポテンシャル計算を含めたハミルトニアン計算の相対性能を示した。この値が1より大きい場合、Z次元拡張がハミルトニアン計算において効果があることを意味している。Z次元拡張のみ、拡張領域に対するデータコピーがステンシル計算の外で必要となっているため、ハミルトニアン計算の性能を比較する必要がある。自動ベクトル化と手動ベクトル化実装ともに、連続領域のみにもかかわらず周期境界領域のデータコピーのコストが高く、ハミルトニアン計算全体では実行時間が増加した。自動ベクトル化実装ではステンシル計算性能が向上したが1.03倍と小さな性能向上にとどまり、手動ベクトル化では0.93倍と性能が低下した。本研究の場合、ハミルトニアン計算全体を考慮するとZ次元拡張の効果は低かったが、一般的なステンシル計算においては、一考の余地がある。

5. 考察

5.1 各プロセッサでの性能

KNCの性能の妥当性について、Xeon CPU との比較およびループラインモデルを用いた期待性能に基づき検討する。本研究では、Ivy-Bridge と Haswell プロセッサをCPUとして取り上げ、Intel コンパイラ 16.0.0 を用いて計測した。各プロセッサでの性能を表 13 に示す。Ivy-Bridge では、KNC での実装を基にAVXの手動ベクトル化実装を行ったが、本研究ではその詳細は省略する。Haswellの場合、手動ベクトル化実装に対し良好な結果となったため自動ベクトル化実装の性能を記載している。KNCがピーク性能比約20%程度であるのに対して、Ivy-Bridge と Haswell ではピーク性能比がそれぞれ57%、44%と高い実効性能を達成している。性能差がキャッシュメモリ性能に大きく影響されているのではないかと仮定のもとに、以下、考察を行う。

まず、ステンシル計算領域である1個の実空間のサイズは 16^3 より64KBである。L1D キャッシュサイズはすべてのプロセッサで32KBであり、これは実空間の半分のサイズであり入りきらないため、L2 キャッシュからL1D キャッシュへのバンド幅が重要と考えられる。ステンシ

表 13 各プロセッサでのステンシル計算性能

Table 13 The performance of the stencil computation on each processor.

Processor	Vectorize	GFLOPS	ピーク性能	Roofline	ピーク比 [%]	期待性能比 [%]
KNC 7110P	Explicit	224.45	1,074	454.19	20.90	49.42
Ivy-B E5-2670v2	Explicit	114.65	200	115.66	57.32	99.13
Haswell E5-2670v3	Compiler	170.74	384	229.82	44.46	74.34
SPARC64 VIIIfx	Compiler	27.20	128	54.66	21.25	49.76

ル計算では、各点に対する係数が 32KB 必要で、書き込みは non-temporal store を行うため 96KB 程度が 1 回の計算で必要となる。しかしながら、スレッドあたりの L2 キャッシュサイズが最小の KNC であっても、そのサイズは 128KB/スレッドであるため、ステンシル計算に必要なデータはほぼ L2 に収まっている。計算領域である電子軌道関数のサイズは、本研究では波数空間格子とバンドをそれぞれ 8^3 , 16 としたので、512MB となり、Ivy-Bridge と Haswell が持つ L3 キャッシュ 25–30MB にも収まらない。しかしながら、ステンシル計算のサイズは 64KB で、L3 キャッシュには 400–480 個の実空間を載せることが可能である。本研究のステンシル計算では、計算の前処理として電子軌道配列から計算用の一時配列に線形コピーを行う。線形コピーは非常に単純かつ連続なメモリアクセスとなるため、1 つの実空間に対しハミルトニアンを計算している間に、次に計算する実空間がプリフェッチされ、L3 キャッシュにすでにロードされていることは十分に考えられる。そのため、L3 キャッシュを持たない KNC および SPARC64 VIIIfx では、DRAM から L2 キャッシュへのバンド幅に律速され、Ivy-Bridge と Haswell では、L3 キャッシュから L2 キャッシュへのバンド幅に律速されると考えられる。

次に、ルーフラインモデルを用いて性能の妥当性について検討する。ルーフラインモデルは、メモリバンド幅を考慮した性能モデルであり、理論ピーク演算性能、理論メモリバンド幅および要求される FLOP/Byte を使い計算性能を見積もる [18]。理論ピーク演算性能 [GFLOPS] を F 、理論メモリバンド幅 [GB/s] を B 、要求 FLOP/Byte を f/b とし以下の式で計算する。

$$\text{Performance} = \frac{f/b}{f/b + F/B} F \quad (1)$$

本研究の 25 点ステンシル計算の FLOP/Byte は約 0.37 である。この式に従い、各プロセッサのルーフラインモデルに基づく期待性能を表 13 中の “Roofline” に示している。理論メモリバンド幅だが、ここでは L2 キャッシュのバンド幅とした。KNC および Ivy-Bridge は L2 キャッシュの Read 性能が 32 Bytes/Cycle で、Haswell の場合 64 Bytes/Cycle となっている。SPARC64 VIIIfx は、仕様書などに公称値の記載がないため、本研究では文献 [19] を参照し 256 GB/s とした。比較すると、Ivy-Bridge では期待性能とほぼ一致

しており、ルーフラインモデルでは十分な値が得られている。KNC と SPARC64 VIIIfx では、期待性能に対しおよそ半分の性能しか得られていない。SPARC64 VIIIfx では様々な最適化によってメモリアクセスの待ち時間を減らしたが、計算時間全体ではおよそ 14% 程度がキャッシュメモリへのアクセス待ち時間となっており、いまだに性能ボトルネックとなっていることが考えられる。KNC では、L2 キャッシュミスによる影響の可能性が高いと考えられる。実空間は 1 個あたり 64KB だが、出力配列で 64KB、電子軌道配列を更新するため 64KB、また各点に対する係数が 32KB ある。合計で 224KB が 1 個の実空間に対するハミルトニアン計算に必要なだが、前述のとおり 1 回のステンシル計算では 96KB 程度が必要で、L2 キャッシュに収まる。電子軌道配列への加算はステンシル計算後毎回行うため、ここで入力配列の一部が L2 キャッシュから追い出され、次のステンシル計算時にキャッシュミスが発生する。そのため、最終的には L2 キャッシュではなく DRAM のバンド幅に律速されるが、L2 キャッシュから追い出されるのは入力配列全体ではなく一部と考えられ、ルーフラインモデルで正確な値を取り出すのが困難である。他の CPU では、スレッドあたり 256KB 以上の L2 キャッシュが利用できるため、L2 キャッシュミスが発生していないと考えられる。

電子軌道配列の更新で利用するならば、ステンシル計算での non-temporal store の利用は、キャッシュ利用の妨げとも考えられる。しかしながら、通常のストア命令を用いるとステンシル計算中に入力配列が L2 キャッシュから追い出される可能性があり、数十 GFLOPS 程度の性能低下が起る。電子軌道配列の更新は線形加算のため、ステンシル計算のように複雑なキャッシュ利用がない。non-temporal store により、ペナルティが比較的低い電子軌道配列の更新中にキャッシュミスのタイミングがずれることで性能向上につながったと考えられる。

5.2 問題依存の最適化

ここで、本研究で行った格子点のサイズや問題依存の最適化について整理する。自動ベクトル化実装に対する最適化は、多くが一般的にステンシル計算の最適化で用いられる手法である。ただし、近傍点ロード最適化については KNC では高い効果が得られているが、Xeon CPU や

表 14 本研究で取り上げたうち条件を要する最適化
Table 14 The constrained optimization in this study.

最適化	適用条件	補足
非アラインメモリアクセス 近傍点のインデックス計算 周期境界領域の拡張	連続方向次元のサイズがベクトル長の倍数 多数のインデックスを求める場合 周期境界条件を持つ	倍精度複素数以外でも適用可能 複雑な計算が必要な場合に効果が期待 同最適化にともなうコストについて検証が必要

SPARC64 VIIIfx では効果がない。

表 14 に、適用条件を持つ最適化について示す。手動ベクトル化では、連続方向の Z 次元のサイズが 4 の倍数、すなわち倍精度複素数ベクトル長の倍数であることを唯一の条件として実装および最適化を行った。この条件は、本研究で述べた中では非アラインメモリアクセス最適化のみが必要としている。同最適化は、連続方向がベクトル長の倍数であることを前提として、Load 命令および Alignr 命令による近傍点データの整形を行っている。多くのステンシル計算では、格子サイズを増加させることは問題の求解精度を向上させることと等しいため、格子サイズを増やしベクトル長の倍数となるように調整することは可能である。また、同最適化で 4×4 の正方行列を生成したが差分の深さが 4 でない場合、深さ N とすると、 $4 \times N$ の行列を生成するように修正し、基本的に同じ方法で、深さ 4 以外のステンシル計算に対応可能である。本研究の対象は倍精度複素数のためベクトル長は 4 になっているが、倍精度実数では 8、単精度では 16 となる。しかしながら、KNC に限らず、対象とするプロセッサのベクトル長と要素のビット幅に合わせれば同様に実装可能で、表 13 であげた Ivy-Bridge の AVX 実装でも演算パターンは KNC と同様である。

周期境界条件を含む複雑な近傍点インデックス計算のベクトル化は、非常に問題依存の最適化である。同最適化は、各次元で正負方向計 8 個の近傍点を必要とすることをふまえ、非連続方向の Y および X 次元のインデックス計 16 点分をベクトル演算でまとめて求める。本研究で取り上げた計算のように、非常に多数の近傍点が必要な場合や、周期境界条件などインデックス計算が複雑な場合、インデックス計算のベクトル化は効果が期待できる。

最後に、周期境界の拡張だが、同最適化は周期境界条件を持つ問題すべてに適用可能である。特に、自動ベクトル化実装では高い効果が期待されるが、KNC では拡張した領域へのデータコピーコストは無視できないため、シミュレーション全体で性能向上が得られるかは留意が必要となる。

5.3 格子点サイズの計算性能への影響

本研究では、実空間に対する逐次ステンシル計算を、波数空間とバンドのサイズだけ OpenMP で並列に計算する。そのため、波数空間とバンドのサイズはアプリケーションの並列粒度に影響し、実空間のサイズは演算性能に大きく

表 15 格子点サイズの計算性能 [GFLOPS] への影響

Table 15 The GFLOPS performance impact of the grid size.

Z	Y	X	Optimized	Blocking
16	16	64	208.65	200.58
16	64	16	133.80	182.57
64	16	16	101.95	126.23
16	16	128	205.54	205.64
16	128	16	48.50	165.48
128	16	16	44.44	43.62
48	48	16	45.55	137.05
48	16	48	148.10	150.49
16	48	48	175.60	187.35

影響する。ここでは、実シミュレーションで用いられている実空間のサイズに対する演算性能への影響について考察する。

実空間のサイズを変化させたときの KNC のステンシル計算性能について表 15 に示す。“Optimized” は本研究で行った最適化実装での性能を示し、“Blocking” は同実装に対し Y-X 次元にブロッキングを適用した場合の性能を示している。この実験では、ある 1 次元のみ 64, 128 と極端に増加させた場合、2 つの次元のサイズを増加させた場合にどのような性能変化が起こるかを測定した。Z と Y 次元どちらかのサイズが増加することによる性能低下が顕著なのに対し、ZY-stride の X 次元は元々 L1 キャッシュに載っておらず、サイズ増加の影響が小さい。また前述のとおり、KNC ではどのサイズでも L2 キャッシュミスが発生し、なかでも最外の X 次元のデータがキャッシュから追い出されている可能性がある。そのため、(16, 16, 128) では (16, 16, 16) に対し軽微な性能低下にとどまっているのではないかと推測している。Z-Y 平面は、本研究の評価で用いたサイズ (16 × 16) では L1 キャッシュに収まっており、Z-Y 平面のサイズが増加することでキャッシュミスが増加し、性能低下につながっていると考えられる。また、Z-Y 平面に対するブロッキングも考えられるが、Z 次元の格子点サイズが 128 でもベクトルループサイズは 32 と非常に小さく、ブロッキングによるオーバーヘッドが高い。我々の実装では、ブロッキング非適用時に対し性能が低下する場合がほとんどであったため、本実験では Z-Y 平面に対するブロッキングを行っていない。Z 次元のサイズ増加に対し、ブロッキングの効果が非常に限定的となっていることが分かる。今回適用したブロッキングは Y と X 次元への

2次元ブロッキングだが、3次元ブロッキングを行った場合でも性能特性に変化は見られなかった。並列化によって、1スレッドに割り当てられる連続方向の次元が他の次元に比べて極端に大きい問題については注意が必要と考えられる。

ブロッキングを行った場合でも、1スレッドで計算する実空間のサイズが増加することによって100 GFLOPS以上の性能低下が見られる。本実験から、1スレッドで計算する実空間のサイズはできる限り小さく、L2 キャッシュには収まることが望ましい。実空間がL2 キャッシュに収まらない程度に巨大な場合、並列処理の最適化としては、OpenMPのnested parallelismを用いて1つの実空間を複数スレッドで計算することが考えられるが、並列化のオーバーヘッドが大きく現実的ではない[20]。アプリケーションの最適化としては、直交座標系に限られている現在の計算コードの仕様を斜交座標に拡張し、より小さい単位セル計算での計算を可能とすることがあげられる。全体としての計算量は変わらないが、波数空間格子が増加するため、十分な並列性を確保しつつ、実空間のサイズを減らすことができ、メニーコアプロセッサに向けた最適化といえる。

5.4 Knights Landing など他のメニーコアプロセッサへの適用

2016年内には、米国の国立エネルギー研究科学計算センター(NERSC)などでKNLを搭載した大規模システムが稼働する予定となっている。また、京コンピュータの後継として現在FX100システムが名古屋大学や核融合研究所で稼働しており、同システムは合計34コアを持つSPARC64 XIfxプロセッサを搭載している。

KNLでは、KNCとは異なりAVX-512がSIMD命令として提供されるため、本研究で実装した手動ベクトル化コードの実装を修正する必要がある。しかしながら、IMCIとAVX-512には不完全ではあるが下位互換性があり、ほとんどの命令は同じフォーマットで利用できる。本研究の実装では、四則演算命令を除外すると13命令を使用しており、そのうち3命令がIMCIとAVX-512でフォーマットが異なる。修正は、単純な命令置換、または数行のインライン関数レベルでの置換にとどまっており、修正コストは非常に小さい。すでに、現在のIMCI実装に対し修正を行い、Intelのエミュレータ[21]上で動作することを確認している。また、AVX-512では機能ごとにサブセットが定義されているが、本研究の実装ではAVX-512対応の全プロセッサで提供されるAVX-512Fのみが必要で、AVX-512対応のプロセッサがあれば性能評価が可能である。

SPARC64 XIfxは、32個の演算コアと2個のアシスタントコアが1つのチップに搭載されているメニーコアプロセッサである[22]。同プロセッサでは、L1Dキャッシュサイズが2倍の64KBとなり、バンド幅も大幅に改善され、

単純な移植でもある程度の性能向上が期待される。また、新しいSIMD命令としてHPC-ACE2を提供し、AVXと同じSIMD長の256-bit SIMD演算が可能となっている。今回、IMCI実装のAVXへの変換は省略したが、HPC-ACE2を用いることで手動ベクトル化実装と高速化が可能と考えている。

6. 結論

本研究では、電子動力学シミュレーションで現れる倍精度複素数の25点ステンシル計算について、KNCへの最適化を行った。まず、アプリケーションの元のターゲットシステムである京コンピュータのプロセッサSPARC64 VIIIfxで、コンパイラによる自動ベクトル化で14.94 GFLOPSから27.20 GFLOPSに性能が改善された。同実装を、KNC上でさらに最適化し、30.06 GFLOPSから130.44 GFLOPSに性能が向上した。さらなる最適化で、KNCで利用できるSIMD命令のIMCIを用いて手動ベクトル化を行った。手動ベクトル化実装では複素数積の展開、非アラインされたメモリアクセスの最適化、インデックス計算のベクトル化を行うことで224.45 GFLOPS、ピーク演算性能比で20.9%を達成した。また、Xeonプロセッサとの比較を行い、Haswellプロセッサに対し1.3倍の性能が得られていることを確認した。HaswellやIvy-Bridgeで高いピーク性能比が得られているのは、L3キャッシュによる恩恵と考えられる。またルーファインモデルを用いて、性能の妥当性や、キャッシュメモリの効果について考察した。本研究で行った問題依存またはサイズ依存の最適化について、他のステンシル計算へ適用する場合の方法などについてまとめ、実空間のサイズが性能に与える影響についても考察した。メニーコアプロセッサを効率的に利用するためには、並列処理最適化だけではなくシミュレーション修正のアプローチが必要となることは、KNCだけでなく、すべてのメニーコアプロセッサ共通の課題である。

最後に、Knights LandingおよびSPARC64 XIfxへの適用について考察した。今後、これらのプロセッサへの適用および性能評価を行うことを予定している。本研究で実装した最適化を含むアプリケーションコードはCOMAおよび京コンピュータに対する最適化実装として、オープンソースソフトウェアとして公開している[23]。

謝辞 本研究の評価環境は、筑波大学計算科学研究センター平成28年度学際共同利用プログラム課題「時間依存密度汎関数理論によるパルス光と物質の相互作用」、HPCI平成28年度「京」一般利用課題「極限的パルス光と物質の相互作用を記述するマルチスケール第一原理計算」、文部科学省ポスト「京」重点課題(7)「次世代の産業を支える新機能デバイス・高性能材料の創成」による。本研究の一部はJST-CREST研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」、研究課題「ポ

ストバスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による。本研究の最適化にあたり、京都大学学術情報メディアセンターの中島浩教授に多くのアドバイスをいただきました。深く感謝申し上げます。

参考文献

- [1] TOP500, available from (<http://www.top500.org/>).
- [2] Heybrock, S., Joó, B., Kalamkar, D.D., Smelyanskiy, M., Vaidyanathan, K., Wettig, T. and Dubey, P.: Lattice QCD with Domain Decomposition on Intel Xeon Phi Co-processors, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14*, IEEE (online), DOI: 10.1109/SC.2014.11 (2014).
- [3] Aprà, E., Klemm, M. and Kowalski, K.: Efficient Implementation of Many-body Quantum Chemical Methods on the Intel Xeon Phi Coprocessor, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14* (online), DOI: 10.1109/SC.2014.60 (2014).
- [4] Nakashima, H.: Manycore challenge in particle-in-cell simulation: How to exploit 1 TFlops peak performance for simulation codes with irregular computation, *Computers & Electrical Engineering*, Vol.46, pp.81-94 (online), DOI: 10.1016/j.compeleceng.2015.03.010 (2015).
- [5] Sato, S.A. and Yabana, K.: Maxwell + TDDFT multi-scale simulation for laser-matter interactions, *J. Adv. Simulat. Sci. Eng.*, Vol.1, No.1, pp.98-110 (2014).
- [6] Schultze, M., Ramasesha, K., Pemmaraju, C.D., Sato, S.A., Whitmore, D., Gandman, A., Prell, J.S., Borja, L.J., Prendergast, D., Yabana, K., Neumark, D.M. and Leone, S.R.: Attosecond band-gap dynamics in silicon, *Science*, Vol.346, No.6215, pp.1348-1352 (online), DOI: 10.1126/science.1260311 (2014).
- [7] RIKEN AICS: K Computer, available from (<http://www.aics.riken.jp/en/k-computer/>).
- [8] 廣川祐太, 朴 泰祐, 佐藤駿丞, 矢花一浩: Xeon Phi クラスタにおける Symmetric 並列実行による電子動力学シミュレーションの性能評価, 情報処理学会研究報告, Vol.2015-HPC-151, No.18 (2015).
- [9] Maruyama, T.: SPARC64(TM) VIIIfx: Fujitsu's New Generation Octo Core Processor for PETA Scale Computing, *HotChips 21* (2009).
- [10] 松田元彦, 丸山直也, 滝沢真一郎: Xeon Phi (Knights Corner) の性能特性とステンシル計算の評価, 情報処理学会研究報告, Vol.2014-HPC-143, No.32 (2014).
- [11] 伊奈拓也, 朝比祐一, 井戸村泰宏: テラフロップス級メモリーコアーキテクチャにおけるステンシル計算の最適化手法の開発, 情報処理学会研究報告, Vol.2015-HPC-152, No.10 (2015).
- [12] 筑波大学計算科学研究センター: COMA, 入手先 (http://www.ccs.tsukuba.ac.jp/files/coma-general/coma_outline.pdf).
- [13] Krishnaiyer, R., Kultursay, E., Chawla, P., Preis, S., Zvezdin, A. and Saito, H.: Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor, *IPDPSW 2013*, pp.1575-1586 (2013).
- [14] Intel: VTune Amplifier XE, available from (<https://software.intel.com/en-us/intel-vtune-amplifier-xe>).
- [15] Takahashi, D.: Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors, *IWIA 2007*, pp.53-59 (2007).
- [16] Hofmann, J., Treibig, J., Hager, G. and Wellein, G.: Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips, *WPMVP'14*, pp.57-64 (2014).
- [17] Andreolli, C.: Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO), available from (<https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso>) (2014).
- [18] Williams, S., Waterman, A. and Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures, *Comm. ACM*, Vol.52, No.4, pp.65-76 (online), DOI: 10.1145/1498765.1498785 (2009).
- [19] 南 一生, 井上俊介, 堤 重信, 前田拓人, 長谷川幸弘, 黒田明義, 寺井優晃, 横川三津夫: 「京」コンピュータにおける疎行列とベクトル積の性能チューニングと性能評価, HPCS2012 ハイパフォーマンスコンピューティングと計算科学シンポジウム, pp.23-31 (2012).
- [20] 李 珍泌, 杉山大輔, 村井 均, 佐藤三久: 軽量スレッドライブラリ Argobots を用いた OpenMP 実装の性能分析と改善, 情報処理学会研究報告, Vol.2016-HPC-153, No.20 (2016).
- [21] Intel Software Development Emulator, available from (<https://software.intel.com/en-us/articles/intel-software-development-emulator>).
- [22] Yoshida, T. et al.: Sparc64 XIfx: Fujitsu's Next-Generation Processor for High-Performance Computing, *HotChips 25* (2014).
- [23] ARTED Github, available from (<https://github.com/ARTED/ARTED>).



廣川 祐太 (学生会員)

1991 年生。2014 年東京工科大学コンピュータサイエンス学部卒業。2016 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了。修士(工学)。同年 4 月より筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士後期課程在籍。メモリーコアプロセッサとアクセラレータ, およびそれらを用いた大規模並列シミュレーションに興味あり。ACM 学生会員。



朴 泰祐 (正会員)

1960年生。1984年慶應義塾大学工学部電気工学科卒業。1990年同大学大学院理工学研究科電気工学専攻後期博士課程修了。工学博士。1988年慶應義塾大学理工学部物理学科助手。1992年筑波大学電子・情報工学系講師, 1995年同助教授, 2004年同大学大学院システム情報工科学研究科助教授, 2005年同教授, 現在に至る。超並列計算機アーキテクチャ, ハイパフォーマンスコンピューティング, クラスタコンピューティング, GPUコンピューティングに関する研究に従事。筑波大学計算科学研究センターにおいて, 超並列計算機CP-PACS, PACS-CS, HA-PACS等の研究開発を行う。2002年および2003年度情報処理学会論文賞, 2011年ACMゴードンベル賞, 2012年度情報処理学会山下記念研究賞各受賞。2015年度情報処理学会フェロー認証。IEEECS, ACM各会員。本会フェロー。



佐藤 駿丞

1990年生。2012年筑波大学理工学群物理学類卒業。2016年同大学大学院数理物質科学研究科物理学専攻博士後期課程修了。博士(理学)。同年4月日本学術振興会特別研究員(筑波大学計算科学研究センター)。第一原理計算を用いた物質科学に関する研究に従事。



矢花 一浩

1960年生。1982年京都大学理学部卒業。1987年同大学大学院理学研究科物理第2専攻修了。理学博士。同年4月日本学術振興会特別研究員(京都大学基礎物理学研究所)。1988年2月新潟大学理学部助手。1994年11月同大学理学部・自然科学研究科助教授。1999年8月筑波大学物理学系助教授。2002年10月同大学院数理物質科学研究科教授。2004年4月同大学計算科学研究センター教授, 現在に至る。計算物質科学, 特に時間依存密度汎関数理論に基づく電子ダイナミクスの第一原理計算の手法開発と応用を行う。日本物理学会, 応用物理学会各会員。